Strong Agent Mobility for Aglets
based on the IBM JikesRVM

Giacomo Cabri, Luca Ferrari, Letizia Leonardi, Raffaele Quitadamo
Dipartimento di Ingegneria dell’Informazione – Università di Modena e Reggio Emilia
Via Vignolese, 905 – 41100 Modena – ITALY
E-mail: { cabri.giacomo, ferrari.luca, leonardi.letizia, quitadamo.raffaele}@unimo.it

ABSTRACT
Mobility enables agents to migrate among several hosts, becoming active entities of networks. Java is today one of the most exploited languages to build mobile agent systems, thanks to its object-oriented support, portability and network facilities. Nevertheless, Java does not support strong mobility, i.e., the mobility of threads along with their execution state; thus developers cannot develop agents as real mobile entities. This paper reports our approach for Java thread strong migration, based on the IBM Jikes Research Virtual Machine, presenting our results and proposing an enrichment of the Aglets mobile agent platform in order to exploit strong agent mobility.

1. INTRODUCTION
Agents are autonomous, proactive, active and social entities able to perform their task without requiring a continue user interaction [22]; thanks to the above features, the agent-oriented paradigm is emerging as a feasible approach to the development of today’s complex software systems [17]. Moreover agents can be mobile, which means they can migrate among different sites/hosts during their execution.

Mobility is an interesting feature for agents, since they are able to move among networks to find out data and information, to perform load balancing activities, and so on. The exploiting of mobile agents can simplify different issues in the design and implementation of applications and can enable developers to quickly build distributed and parallel systems.

Thanks to its portability and network facilities, Java is today the most exploited language to develop mobile agents, and in fact several Java-based Mobile Agent Platforms (MAP) exist [16, 2, 26]. Unfortunately, current standard Java Virtual Machines (JVMs) do not support thread migration natively.

With regard to mobility, we distinguish strong mobility, which enables the migration of code, data and execution state of execution units (for instance, threads), from weak mobility, which migrates only code and data [14].

From the complexity point of view, weak mobility is quite simple to implement using well-established techniques like network class

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SAC’06, April, 23-27, 2006, Dijon, France.
Copyright 2006 ACM 1-59593-108-2/06/0004...$5.00

loading or object serialization [25]. However, weak mobility systems, by definition, discard the execution state across migration and hence, if the application requires the ability to retain the thread of control, extra programming effort is required in order to manually save the execution state. The transparency of the migration offered by strong mobility systems has instead a twofold advantage: it reduces the migration programming effort to the invocation of a single operation (e.g. migrate() method), and requires a size of the migrated code smaller because it does not add artificial code. Despite these advantages, most of the mobile agent systems support only weak mobility and the reason lies mainly in the complexity issues of strong mobility and in the insufficient support of existing JVMs to deal with the execution state. It is a common idea that strong mobility should be convenient only in load balancing contexts or when thread persistence is needed to build fault-tolerant applications [7]. Moreover, a weakly mobile system gives the programmer more control over the amount of state that has to be transferred, while an agent using strong migration may bring unnecessary state, increasing the size of the serialized data.

To overcome many of these limitations, this paper proposes an approach to support strong thread migration for Java MAPs, based on the IBM JikesRVM [4], which is a Java virtual machine with very interesting features. Our approach significantly differs from other proposals, since it requires neither any modification to the JVM, nor it exploits any pre-processing, but it simply defines an appropriate Java library.

The paper is organized as follows: section 2 presents the state of the art, explaining the existing approaches and pointing out their limitations. Section 3 introduces the features of JikesRVM and explains how they can be exploited to build a library for supporting strong mobility. Section 4 presents the Aglets platform and shows how strong mobility can be designed and implemented in such a platform by using our strong mobility library. Finally, Section 5 reports some first performance evaluation tests and Section 6 concludes the paper.

2. STATE OF THE ART
The approach presented in this paper aims at implementing strong thread migration in Java, which is not a new idea. Several approaches have been proposed so far and they can be, typically, split into two categories, depending on the fact that they require to modify the JVM (JVM-level approach) to support an advanced thread management or exploit some kind of bytecode instrumentation (Application-level approach) to track the execution state of each thread.

Approaches that modify the JVM (such as Sumatra [1], ITS [8], Merpati [27], Jessica2 [31], JavaThread [7] and NOMADS [28]) often introduce the problem of the management of the virtual
machine itself. First of all, it is worth noting that they are applied to JVMs that are at least one (or even more) version older than the SUN production one. Second, the adoption of a modified JVM can introduce problems of trust and security bugs. Third, virtual machines are usually written in a language different from Java (e.g., C++), thus suffering from portability problems.

Instead, approaches that exploit bytecode manipulation (e.g., JavaGoX [24], Brakes [29] and [13]), even if based on a pure Java technique (and thus really portable), do not provide a complete thread management and suffer from problems related to performances. In fact, the idea of these approaches is to transparently place a few control instructions, similar to recovery-points, that allow a thread to deactivate itself once it has reached one of them. Recovery-points are quite similar to entry points used in most Java MAPs (i.e., methods that are executed when an agent is reactivated at the destination host), even if the former ones enable a finer grain control than entry points. Un22ily, a thread cannot deactivate (or reactivate) itself outside of these recovery-points, which are also not customizable, thus a thread cannot really suspend itself in an arbitrary point of the computation. Moreover, the use of bytecode manipulation produces low performances, thus these techniques are not appropriate for those applications where speed represents a key requirement.

In general, all existing strongly mobile systems have to deal with the problem of locating object references when they want to migrate a thread with all its set of stack-referenced objects. The adopted solution is to force the use of some “type inference” mechanism [7, 31], either at execution or at compilation time, thus introducing a significant performance overhead in threads execution.

Starting from the above considerations, we have decided to design and implement a strong thread migration system able to overcome all the problems of the above-explained approaches. In particular, it is written entirely in Java, thus portable as much as possible, and it grants good performances even without modifying the JVM. In fact, every single component of the migration system has been designed and developed to be used as a normal Java library, without requiring rebuilding, changing or patching the virtual machine, in specific, the IBM JikesRVM. Programmers and users do not have to download a modified, untrustworthy, version of JikesRVM, but can import the implemented mobility package into their code and execute it on their own copy of JikesRVM. Therefore, our JikesRVM-based approach can be classified as a midway approach between the above-mentioned JVM-level and Application-level approaches. Other midway approaches [15] exploit the JPDA (Java Platform Debugger Architecture) that allows debuggers to access and modify runtime information of running Java applications. The JPDA can be used to capture and restore the state of a running program, obtaining a transparent migration of mobile agents in Java, although it suffers from some performance degradation due to the debugger intrusion.

3. FROM WEAK TO STRONG MOBILITY:
A JIKESRVM-BASED APPROACH

Recently, an innovative project is drawing researcher’s attention to the benefits that a virtual machine written in the Java language can offer. The main features of this open-source project, called JikesRVM [4], are outlined in the following subsection: for the sake of brevity, we will focus on those aspects that make JikesRVM an ideal execution environment for strongly mobile agents, overcoming the drawbacks and the limitations of many existing solutions.

JikesRVM began life in 1997 at IBM T. J. Watson Research Center as a project with two main design goals: supporting high performance Java servers and providing a flexible research platform “where novel VM ideas can be explored, tested and evaluated” [3]. JikesRVM is almost totally written in the Java language, but with great care to achieving maximum performance and scalability exploiting as much as possible the target architecture’s peculiarities. The all-in-Java philosophy of this VM makes very easy for researchers to manipulate or extend its functionalities. Furthermore, JikesRVM source code can be built, with a prior custom compilation, both on IA32 and on PPC platforms [18], but the bulk of the runtime is made up of Java objects portable across different architectures.

The first step toward the development of our MAP based on JikesRVM has been the implementation of the strong Java thread mobility. Threads embody concurrent flows of execution within an instance of the JVM and are represented by java.lang.Thread objects [21], used by the Java programmers disregarding any knowledge of their underlying physical implementation. In JikesRVM, threads are full-fledged Java objects and are designed explicitly to be as lightweight as possible [3]. As well as many server applications need to create new threads for each incoming request, a Mobile Agent Platform has similar requirements since thousands of agents may request to execute within it. While some JVMs adopted the so-called native-thread model (i.e. the threads are scheduled by the operating system that is hosting the virtual machine), JikesRVM designers chose the green-thread model [23]: Java threads are hosted by the same operating-system thread, implemented by so-called virtual processor, through an object of class VM_Processor [5]. Each virtual processor manages the scheduling of its virtual threads (i.e., Java threads), represented by objects of the class VM_Thread. The scheduling of virtual threads was defined quasi-preemptive, since it is driven by the JikesRVM compiler. What happens is that the compiler introduces, within each compiled method body, special code (yield points) that causes the thread to request its virtual processor if it can continue the execution or not. If the virtual processor grants the execution, the virtual thread continues until a new yield point is reached, otherwise it suspends itself so that the virtual processor can execute another virtual thread.

The choice of using virtual processors not only allows JikesRVM to reduce the number of threads the operating system is in charge of, but also allows it to perform an efficient and well-controlled thread-switch. As a consequence, this allows elegantly addressing the problem of precisely locating object references when a garbage collection occurs. JikesRVM uses type-accurate collectors [30] that build the so-called reference maps automatically at compile-time, unlike conservative collectors, which attempt somehow to infer whether a stack word is a reference or not. These reference maps are periodical snapshots of the situation of references in each method frame. The tracks of object references used to speed up the JikesRVM type-accurate garbage collectors can be exploited by MAP designers to collect stack-referenced objects for strong thread migration. This eliminates the need for “type inference” mechanisms required by existing strongly mobile systems.

In general, many JVMs do not permit the programmer to access the execution state (i.e. the stack and the context registers), in order to enforce the security model of the Java language. As a
As many other Java MAPs, Aglets exploits weak mobility, that means, from a programming point of view, that each time an agent will be posted to the aglet and this thread will invoke the appropriate methods transparently invoked by the platform during the agent execution in the destination platform, allowing possibly a clean removal of the dangerous actions by malicious (or bugged) aglets do not affect the stability of our platform, allowing possibly a clean removal of the dangerous agent without the need of a MAP reboot.

As many other Java MAPs, Aglets exploits weak mobility, that means, from a programming point of view, that each time an agent will be migrated. The presented features of JikesRVM allow the addition of strong thread migration, without modifying the virtual machine, but simply extending it. The entire system is available as a library comprised in a Java package that can be imported as usual into the application code. This means that the implemented JikesRVM extension does not affect the performance of other applications, since no permanent modifications have been made to the VM itself.

### 4. STRONG MOBILITY IN AGLETS

In this section, we explain how the JikesRVM has been exploited to achieve strong mobility in Aglets. In the first subsection we present the current implementation of Aglets, based on weak mobility. In the further subsections we detail the needed additions and modifications, results of our research.

#### 4.1 Overview Of The Aglets Workbench

The Aglets Workbench [2] is a project originally developed by the IBM Tokyo Research Laboratory with the aim of producing a platform for the development of mobile agent based applications by means of a 100% Java library. The Aglets Workbench provides developer with applet-like APIs [19], thus creating a mobile agent (called Aglet) is a quite straightforward task. It suffices to inherit from the base class Aglet and to override some methods transparently invoked by the platform during the agent life. Weak mobility is provided through the Java serialization mechanism, and a specific agent transfer protocol (ATP) has been built on top of such mechanism [20]. Each Aglet can exploit the special method `dispatch()` to move to another host.

As many other Java MAPs, Aglets exploits weak mobility, that means, from a programming point of view, that each time an agent is resumed at a destination machine, its execution restarts from a defined entry point, that is the `run()` method call. Due to this, dealing with migrations, the code will appear like the one shown in the simple example of Figure 1. There, in case of a single migration, the migrated flag is used to select a code branch for the execution either on the source or destination machine.

The code of Figure 1 is just a simple example, but more complex agents follow the same programming style. The point in all such cases is that with weak mobility, which is the one provided by the Java language and the most existing MAPs, it is as the code routinely performs rollbacks. In fact, looking at the code in Figure 1, it is clear how, after a successful `dispatch()` method call that causes the agent migration, the code does not continue its execution in the `run()` method from that point. Instead, the code restarts from the beginning of the `run()` method (on the destination machine, of course), and thus there is a code rollback. The fact that an agent restarts its execution always from a defined entry point, could produce awkward solutions, forcing the developer to use flags and other indicators to take care of the host the agent is currently running on.

```java
public class MyAgent extends Aglet{
    protected boolean migrated = false; // indicates if the agent has moved yet
    public void run(){
        if( ! migrated ){
            // things to do before the migration
            migrated = true;
            try{
                dispatch(new URL("atp://nexthost.unimore.it");
            }catch(Exception e){ migrated = false; }
        }
        else{
            // things to do on the destination host
            // ...
        }
    }
}
```

Figure 1. An example of Aglet with a single migration.

#### 4.2 Implementing Strong Mobility

In Section 3 we have presented the innovative features of JikesRVM that can be exploited to strongly migrate threads. Now we apply these features to the Aglets to realize the idea of an Aglet as a strong migrable thread. Instead of using one of the pre-created threads to execute methods of the aglets, JikesRVM makes feasible to have a single independent thread for each aglet. As already mentioned, this is possible because of the lightweight implementation of Java threads in that JVM, being targeted to server architectures, where scalability and performance are key requirements. Furthermore, having a separate thread for each aglet ensures a high level of isolation between agents: consider, for example, the case where an agent wants to sleep for some time, without being deactivated (i.e. serialized on the hard disk). Using the classical `sleep()` method on the Java.lang.Thread object will produce strange effects on the current Aglets implementation platform (such as locking the message passing mechanism). These shortcomings are due to the thread sharing among multiple agents through the pool of threads. Instead, potentially dangerous actions by malicious (or bugged) aglets do not affect the stability of our platform, allowing possibly a clean removal of the dangerous agent without the need of a MAP reboot.

There is only a thread responsible for handling the messages posted to the aglet and this thread will invoke the appropriate handler function to perform the necessary actions in response to the delivered message. In the official Aglets framework, the
thread running into the handler function cannot be interrupted asynchronously by a migration request, notified by another thread by means of the *dispatch* message. In our prototype, the *dispatch* message has the effect of suspending the execution of the function (in particular, the handler of the *run* message, i.e. the *run()* method) and migrating the aglet to the designated host. The *OnDispatching()* handler method is executed to allow preparatory actions to be done, but the current execution stack is preserved, together with local variables, stack operands and method parameters. There is no more need for saving intermediate results into serializable fields or structuring the code with entry points from which the agent execution is restarted each time it arrives at a new host. Referring to the code example of Figure 1, the adoption of strong thread mobility overtakes the mentioned drawbacks, since the code restarts at the destination machine from the same point it stopped at the source one. Thus the code shown in Figure 1 becomes the one of Figure 2. As readers can see, the code is simpler (no flags and branches are required) and shorter than the previous one.

```java
class MyAgent extends Aglet {  
   public void run() {  
      // things to do before the migration
      try {  
         migrate(new URL("http://nexthost.unimore.it"));  
      } catch (Exception e) { ... }  
      // things to do after migration
   }
}
```

**Figure 2. An example of Aglet code using our approach.**

This kind of message-driven strong mobility is achieved serializing the aglet object and its fields but also appending a sequence of stack frames, representing the state of the execution at the time of the suspension. These frames are extracted from the stack of the thread with the aid of the JikesRVM-integrated OSR facility: walking back the stack, from the lastly called method to the first one (i.e. the message handler function), a portable bytecode-level representation of each frame (we will call this a MobileFrame) is retrieved, as shown in Figure 3.

The MobileFrame object, modelled on the OSR scope descriptor [12], contains the necessary data, required to correctly re-install the physical frame on another remote JikesRVM instance. The OSR scope descriptor (and the MobileFrame too) is independent of the underlying platform where JikesRVM was build, being it a Linux/IA32 machine or a MacOS/PPC one: for instance, the bytecode index is used as the program counter for each method activation in order to guarantee maximum portability among different architectures. Once again, the mapping between bytecode index and machine code offset into the compiled method bodies is maintained internally by the JikesRVM runtime compiler: thus, no further bytecode analysis is needed at migration time. In addition, the MobileFrame object represents a packaged little slice of the thread’s dataspace [14], with all the serializable objects referenced by local variables, stack operands and parameters.

In a few key points, the de-serialization process involves

1. reading the aglet object from the network stream into the memory;
2. creating a new thread for this aglet or acquiring an existing one, if available;
3. notifying this thread of the arrival event and suspending its execution;
4. injecting on the fly all the frames into its stack;
5. resuming the execution of the thread/aglet transparently.

The migrated aglet will be, by default, destroyed in the source JVM and its associated thread added back to the thread pool, for a possible future reuse. Nevertheless, the *dispose* message can be explicitly intercepted by the programmer so that the aglet can continue executing, thus realizing a form of “agent cloning”.

![Figure 3. The MobileFrame object.](image)

### PERFORMANCE AND DISCUSSIONS

At the current stage of our research, the thread serialization mechanism, integrated into the Aglets framework, has been successfully tested, focusing mainly on the state capturing and restoring of the threads executing the aglets.

First of all, we made some first performance tests to discover possible bottlenecks and evaluate the cost of each migration phase. The times measured are expressed in seconds and are average values computed across multiple runs, on a Pentium IV 3.4Ghz with 1GB RAM on JikesRVM release 2.4.1. We tested the serialization with increasing stack sizes (5, 15 and 25 frames) and found a very graceful time degradation. These times are conceptually divided into two tables, where Table 1 refers to the thread serialization process, while Table 2 refers to the symmetrical de-serialization process at the arrival host.

#### Table 1. Evaluated times for thread serialization (sec.)

<table>
<thead>
<tr>
<th></th>
<th>5 frames</th>
<th>15 frames</th>
<th>25 frames</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSR Frame capturing</td>
<td>1.78E-5</td>
<td>1.89E-5</td>
<td>1.96E-5</td>
</tr>
<tr>
<td>State building</td>
<td>3.44E-5</td>
<td>3.75E-5</td>
<td>3.43E-5</td>
</tr>
<tr>
<td>Pure serialization</td>
<td>2.49E-3</td>
<td>7.32E-3</td>
<td>1.50E-2</td>
</tr>
<tr>
<td>Overall times</td>
<td>2.54E-3</td>
<td>7.38E-3</td>
<td>1.51E-2</td>
</tr>
</tbody>
</table>

#### Table 2. Evaluated times for thread rebuilding (sec.)

<table>
<thead>
<tr>
<th></th>
<th>5 frames</th>
<th>15 frames</th>
<th>25 frames</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pure de-serialization</td>
<td>4.46E-3</td>
<td>5.33E-3</td>
<td>7.06E-3</td>
</tr>
<tr>
<td>State rebuilding</td>
<td>5.45E-4</td>
<td>5.27E-4</td>
<td>5.06E-4</td>
</tr>
<tr>
<td>Stack installation</td>
<td>1.53E-3</td>
<td>1.60E-3</td>
<td>1.71E-3</td>
</tr>
<tr>
<td>Overall times</td>
<td>6.54E-3</td>
<td>7.46E-3</td>
<td>9.28E-3</td>
</tr>
</tbody>
</table>
Considering how these times are partitioned among the different phases of the thread serialization, we can see that the bulk of the time is wasted in the pure Java serialization of the captured state, while the frame extraction mechanism (i.e., the core of our entire facility) has very short times instead. The same bottleneck due the Java serialization may be observed in the de-serialization of the thread. In the latter case, however, we have an additional overhead in the stack installation phase, since the system has often to create a new thread and compile the methods for the injected frames.

Furthermore, the developed prototype has some limitations that will be dealt with in the future: the first one is about the kind of supported compilers. JikesRVM provides three compilers, designed to achieve different levels of code optimization: a baseline, a quick and an optimizing compiler [5]. The JikesRVM OSR mechanism can capture scope descriptors for those methods compiled by optimized compilers, but this requires maintaining additional structures to cope with parameters allocated into registers, inlined methods and other challenging optimization techniques [12]. Currently, JikesRVM designers allow OSR to occur only at yield points (see section 3) and this implies that not all the optimized frames in the stack have their maps updated. Therefore, our prototype can actually migrate only baseline compiled methods.

Other interesting issues will rise when the prototype will become a full-fledged MAP but, for space limitations, only the most relevant will be discussed in this section. Some issues are about resource relocation strategies [14]: file system object, GUI elements and so forth. Research in the field of mobile code [11] demonstrated that flexibility is crucial when planning dataspace relocation policies. In our prototype, objects are encapsulated into special containers, which realize customizable relocation strategies for the object contained. The idea is to define a limited number of containers, for the most frequent cases (e.g., relocation of a file or a socket with network references, containers for static objects such as System.out, System.in, etc.), while letting the user define her own containers with the required serialization policies and register them for the desired objects.

In addition, we are experimenting with a version capable of migrating also multi-threaded agents, with multi-dependencies and references. In a few words, when a dispatch is notified to the multi-threaded agent, all threads in the group have to reach a synchronization barrier and, only when this condition is met, the entire thread group is serialized and resumed at destination synchronously.

Thread serialization very often implies coping with issues related also to concurrency: a migration can occur when the object is running in a synchronized block or method, but serialization does not preserve such additional object state. Monitors in JikesRVM are implemented by means of two kind of object locks [3]: thin locks, stored into the object header and useful when there is no contention for the object; they are very fast but have to be inflated, becoming thick locks, when a contention is detected. Thick locks are JikesRVM thread-safe objects, with their own queues for those threads blocked outside the monitor and for those ones that invoked a wait (...). Thin locks have been easily relocated with a proper "lock container", that encapsulates the object and its locking information, to be restored in the object’s header at the destination. Thick locks are more challenging and we are working at the definition of the policies required to make them portable, especially when they involve multiple threads in the agent.

6. CONCLUSIONS AND FUTURE WORK
This paper has introduced our approach to support Java thread strong mobility based on the IBM JikesRVM virtual machine, and has outlined how this mechanism is being fully integrated in the Aglets Mobile Agent Platform. Thanks to the support to thread serialization, agents will be simpler in terms of code, and, at the same time, the code will be easier to read.

Our approach represents an extension of JikesRVM but does not change any part of this JVM. Rather, it exploits some interesting facilities provided by that JVM to avoid many of the drawbacks of the presented solutions. In particular, OSR facility allowed us to capture the state (i.e., method frames) in a very portable (i.e., bytecode-level) format. Thanks to the scheduling policy of the JikesRVM, which enables the support of thousands of Java threads, our approach will keep the thread management more lightweight, experimenting the possibility of having one thread for each agent, which is not possible in the current implementation of Aglets. Our JikesRVM-based migration library will enrich the Aglets framework with strong mobility benefits.

With regard to future work, we will perform a comparison test between the current Aglets release (with weak mobility) and our JikesRVM-based version (with strong mobility). This comparison will be performed also under critical conditions (such as a large number of agents). In any case, from the first results reported in section 5, we can draw the conclusion that the prototype can be further optimized with respect to the Java serialization aspect. In fact, even if weak mobility of the current Aglets imposes a certain serialization cost, our strong mobility approach implies more thread state data to be serialized. Then, we have to carefully optimize the internal structure of the thread state, improving the serialization time.

7. ACKNOWLEDGEMENTS
Work supported by the Italian MIUR and CNR within the project "IS-MANET, Infrastructures for Mobile ad-hoc Networks" and by the European Community within the project "CASCADAS".

8. REFERENCES


